RATIONALIZED APL

Kenneth E. Iverson
I.P. Sharp Associates
Toronto

This paper is intended as a companion to **Rationalizing APL** [1], providing loose definitions and numerous examples of use of the functions and operators defined more formally therein. Either paper may be read first, although it is probably best to start with this one, making concurrent reference to the other as desired. The ideas introduced in Sections B to E of [1] (Name Assignment, Function Valence, Function Definition, and Syntax and Order of Execution) will be treated only indirectly by using them in the discussion of other topics.

A final section compares APL2 [2] with a restricted subset of the proposals in [1].

The following functions will be used in examples:

Enclose `<` — $\rho\rho{<}A \leftrightarrow 0$, and $>{<}A \leftrightarrow A$, and $<A$ is called a **non-simple** array.

Disclose `>` — Inverse of `<`, but applies to each element of its argument.

Link `⊃` —
$\supset\omega \leftrightarrow <\omega$ if ω is simple
$\leftrightarrow \omega$ if ω is non-simple
$\alpha\supset\omega \leftrightarrow (<\alpha),\supset\omega$

As in [1], we will use *A*0 to denote a scalar, *A*1 to denote a vector, and so on.

The examples shown in this paper were executed by the function *APL* in workspace 3621609 *MODEL*. 0-origin indexing is assumed thoughout, and infinity is denoted by the overbar ¯.

FUNCTION RANKS AND DISPOSITION OF AXES

Function rank is the most important notion needed to provide a simple and systematic basis for the uniform treatment of all "mixed" or non-scalar functions. It is also relatively difficult to grasp: although the rules for rank are simple, their consequences are neither obvious nor inconsequential, and the reader unfamiliar with function rank should carefully study the simple examples of its use, in preparation for the more complex

and more rewarding uses in dyadic functions, and in conjunction with operators such as composition and dual.

If F has rank R, then $F\omega$ is determined by applying F to each of the "cells" of shape $(-R)\uparrow\rho\omega$, producing a common shape S for each, and assembling the whole into a result of shape $((-R)\downarrow\rho\omega),S$. In other words, the argument ω is treated as an array of **outer shape** $(-R)\downarrow\rho\omega$ of cells of shape $(-R)\uparrow\rho\omega$, and the (common) shape of the result of applying F to any cell is placed last in the shape of the overall result.

Every function has an assigned rank. Thus > has rank 0 (i.e., applies to each scalar), and < has **infinite**, or **unbounded** rank, applying to its entire argument. For example:

```
    A←1 2 3 ⊃ 4 5 6
    A
|1 2 3| |4 5 6|
    B←>A
    B
1 2 3
4 5 6
    ρB
2 3
    <B
|1 2 3|
|4 5 6|
```

The **rank** operator ⍤ produces a function of rank specified by its right argument. Thus:

```
    <⍤1 B
|1 2 3| |4 5 6|
```

Moreover, the ravel function has infinite rank and:

```
    C←2 3 4⍴⍳24
    ,C
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
    ,⍤2 C
 0  1  2  3  4  5  6  7  8  9 10 11
12 13 14 15 16 17 18 19 20 21 22 23

    >,<⍤1 C
 0  1  2  3
 4  5  6  7
 8  9 10 11
12 13 14 15
16 17 18 19
20 21 22 23
```

A negative argument of the rank operator specifies the **complementary** rank, that is the rank of the outer shape. For example:

```
    ,⍤¯1 C
 0  1  2  3  4  5  6  7  8  9 10 11
12 13 14 15 16 17 18 19 20 21 22 23
```

In some instances, a primitive in conventional APL extends usefully by simply making an explicit assignment of the ranks implicit in its definition. For example, ⌹ is presently limited to rank 2 arguments, but extends usefully according to the general rules. More interestingly, ⍋ is defined on vectors, and if assigned rank 1, can be used in the manner illustrated below:

```
    ⍋CH←⍒2 2⍴⍳4
0 1
2 3
```

THE TRANSPOSE OPERATOR

It is sometimes necessary to move certain axes of the argument before applying a function. Although the dyadic transpose function can perform this, it is much more convenient to use the transpose operator defined as follows:

F⍤⍥*K*⍵ Applies *F* after moving axes *K* to the rear.

K⍤⍥*F*⍵ Applies *F* after moving axes *K* to the front.

For example:

```
    ,⍤⍥0 C
0 12 1 13 2 14 3 15 4 16 5 17 6 18 7 19 8 20 9 21 10 22 11 23

    1 2⍤⍥, C
0 12 1 13 2 14 3 15 4 16 5 17 6 18 7 19 8 20 9 21 10 22 11 23

    ,⍤2⍤⍥0 C
 0 12  1 13  2 14  3 15
 4 16  5 17  6 18  7 19
 8 20  9 21 10 22 11 23

    <⍤2⍤⍥0 C
| 0 12| | 4 16| | 8 20|
| 1 13| | 5 17| | 9 21|
| 2 14| | 6 18| |10 22|
| 3 15| | 7 19| |11 23|
```

The last two examples above involve a sequence of operators, and it should be noted that an operator has "long left scope", applying to the result of the entire operator sequence to its left. Thus, ,⍤2⍤0 is equivalent to (,⍤2)⍤0, not to ,⍤(2⍤0). This may be emphasized by assigning a name to the intermediate (function) result produced:

```
    R2←,⍤2
    R2 C
 0  1  2  3  4  5  6  7  8  9 10 11
12 13 14 15 16 17 18 19 20 21 22 23
    R2⍤0 C
 0 12  1 13  2 14  3 15
 4 16  5 17  6 18  7 19
 8 20  9 21 10 22 11 23
```

For transpositions on an argument of arbitrary rank, the transpose operator used in conjunction with the identity function ⊢ can often be much more convenient than the transpose function. For example, to move axes *K* to the rear:

```
⊢⍤K⍵ ↔ (⍋((~A∈K)/A←⍳⍴⍴⍵),K)⍉⍵
```

DYADIC FUNCTIONS

If the function *G*←*F*⍤*R* is to be applied dyadically as well as monadically (the only cases addressed in the preceding sections), then it is necessary that *R* specify three independent ranks, the monadic, the left, and the right. The general argument *R* is therefore a three-element vector that specifies the ranks in the order just indicated. Moreover, *R* is extended by reshape if necessary, so that *F*⍤*R* ↔ *F*⍤3⍴*R*.

The following example shows the behaviour of a useful catenation function that has differing left and right ranks:

```
    G←,⍤¯ 0 1
    A←2 3⍴⍳6
    7 8 G A
7 0 1 2
8 3 4 5
    7 8 ∘.G A
7 0 1 2
7 3 4 5

8 0 1 2
8 3 4 5
    H←∘.G
    ⍴B←3 4 5 H 7 8 H A
3 2 2 5
```

```
    0 0{B
3 7 0 1 2
3 7 3 4 5
```

If $H \leftarrow F\ddot{\circ}R$ is a function of higher rank than F, then F and H may be indistinguishable when used in the expressions $\alpha F\omega$ and $\alpha H\omega$, but may produce quite different results under the application of an operator. More precisely, if I and J are non-negative scalars and if the rank of F is $3\rho I$, then $H \leftarrow F\ddot{\circ}(I+J)$ agrees with F in direct application, but may differ from it under application of an operator.

For example, the rank of the scalar function × is 0, and:

```
    T1←×¨1
    A←2 3⍴⍳6
    (A×A)≡A T1 A
1
    ⍴A∘.×A
2 3 2 3
    A∘.T1 A
0  1  4
0  4 10

0  4 10
9 16 25
```

The left brace denotes a dyadic indexing function called **from**. Its left and right ranks are 1 and ¯, and:

```
    1 2 3{A3 ↔ A3[1;2;3]
```

For example:

```
    A←2 3 4⍴⍳24
    1 2 3{A
23
```

If the left argument of { is of rank greater than 1, then the normal rules of extension apply, giving the useful "scattered" indexing:

```
    I←?4 3⍴2
    I
1 1 0
0 0 0
1 1 1
1 1 0
    I{A
16 0 17 16
```

If I is a non-simple vector (i.e., including one or more elements produced by the enclose function <), then the disclosed elements of I apply as follows:

```
I{A3 ↔ A3[>I[0];>I[1];>I[2]]
```

For example, using the **link** function ⊃ (that encloses its left argument and catenates it to the right, first enclosing the right if it is simple):

```
    I←1⊃1 2⊃0 3
    I{A
16 19
20 23
```

This generalization may also be written as:

```
α{ω ↔ ({α){ω
```

using the monadic case of {, the **cartesian product** discussed in the following section. For example:

```
    {I
1 1 0
1 1 3

1 2 0
1 2 3
    ({I){A
16 19
20 23
```

THE CARTESIAN PRODUCT

The asymmetric catenation function G←,⍤¯1 0 1 illustrated earlier can be used to add a unit final axis. Thus:

```
    8 9 ∘.G ''
8
9
    ⍴8 9 ∘.G ''
2 1
```

It can also be used to form a cartesian product of arrays. For example:

```
    C←2+B←2+A←1 2
    R←A ∘.G B ∘.G C ∘.G ''
    R
1 3 5
```

```
1 3 6

1 4 5
1 4 6


2 3 5
2 3 6

2 4 5
2 4 6
```

Finally, if *I* is the vector *I←A⊃B⊃C*, then the reduction of *I*,∘ by ∘.*G*¨> produces the enclosed cartesian product of the elements of *I*. Thus:

```
    I←A⊃B⊃C
    R≡> ∘.G¨>/I,∘
1
```

The operator ¨ (**dual**) used in the foregoing expression is defined by:

$$F¨G\omega \leftrightarrow G\subset F\ G\ \omega$$
$$\alpha F¨G\omega \leftrightarrow G\subset (G\alpha)\ F\ G\ \ \omega$$

where ⊂ (**con**) denotes the inverse operator. The effect of *F*¨>*ω* is to apply *F* to (the disclose of) each of the ×/ρ*ω* enclosed elements of *ω*, and to enclose each result.

The cartesian product function will be defined as the monadic case of {. Formally:

```
{ω ↔ > ∘.(,⍤¯ 0 1)¨>/(<⍤>ω),∘
```

Its use in the definition of the dyadic case was shown in the preceding section, namely:

```
α{ω ↔ ({α){ω
```

Since { applied to a simple vector yields the vector unchanged, the definition applies (trivially) to a simple vector left argument as well as to a vector of enclosed elements. Note that it is the case of a simple argument that required the expression (<⍤>*ω*) instead of *ω* in the definition of {*ω* above.

In [1], the definition of α{*ω* is extended further to allow **abbreviated, negative,** and **complementary** indexing as well. For example:

```
    A←2 3 4⍴⍳24
```

```
    1 2{A
20 21 22 23

    1 ¯1{A
20 21 22 23
    I←1⊃(<2)⊃0 3
    I
|1| ||2|| |0 3|

    I{A
12 15
16 19
```

Because an element of the form <<'' will select **all** along the corresponding axis, an expression such as ((3ρ<<''),<K){A⍤4 can be used to select an index K along the last axis. However, if it is also desirable to bring the last axis to the front, it is more convenient to use the "display" function D, defined and illustrated below:

```
    D←¯1⍤{
    1 D 3 3 3ρ⍳27
 1  4  7
10 13 16
19 22 25
    0 D 3 3 1ρ⍳9
0 1 2
3 4 5
6 7 8
```

The last example above shows the production of an easily readable display of an array with a final unit axis. This will be used later in the discussion of permutations.

OPERATORS ON NON-SCALAR FUNCTIONS

Some examples of the application of operators (such as the outer product and the rank) have already appeared in earlier sections. We will here provide further examples involving the reduction operators / and ⌿, as well as the scans \ and ⍀, which introduce supernumerary axes.

The expression F⌿ω splits ω along its leading axis, and applies F between the resulting subarrays. Thus, if A←?3 4 5ρ9 then

```
    F⌿A ↔ (0{A) F (1{A) F 2{A
```

For example:

```
    A←3 3⍴⍳9
    ∘.×⌿A
 0  0  0
 0  0  0
 0  0  0

18 21 24
24 28 32
30 35 40

36 42 48
48 56 64
60 70 80

    P←3 3 1⍴1 0 2 2 0 1 2 1 0
    D←¯1⍤{
    0 D P
1 0 2
2 0 1
2 1 0
    ,{⌿P
2 0 1
```

The last result above is the "product of the permutations" represented by the rows of *P*.

The other reduction (*F/*) splits along the last axis. For example:

```
    A←I∘.×10*I←⍳3
    A
  0   0   0
  1  10 100
  2  20 200
    ⍴R←∘.(,⍤0)/A
3 3 3 2 2
    1 2 { R
  1  20
  1   0

  1  20
  1 100

  1  20
  1 200
```

A scan of the form *F⍀ω* applies *F⌿ω* to each of a set of subarrays *(,⍤0⍳K){ω*, for each *K∈1+⍳1↑⍴ω*, and the supernumerary axis (of length *1↑⍴ω*) is placed **before** the axes contributed by the individual results. This will be illustrated in the scan *{⍀P*,

where *P* is the permutation matrix used earlier to illustrate the reduction {⌿:

```
    P←3 3 1⍴1 0 2 2 0 1 2 1 0
    0 (D←⍉1⌽{) P
1 0 2
2 0 1
2 1 0
```

The subarrays involved in the scan {⍀ may be produced and displayed as follows:

```
    G←,⍤0⍤1

    0 D (G 1){P
1 0 2

    0 D (G 2){P
1 0 2
2 0 1

    0 D (G 3){P
1 0 2
2 0 1
2 1 0
```

The scan {⍀*P* therefore gives all of the permutations produced by reductions over these subarrays:

```
    0 D {⍀P
1 0 2
0 2 1
2 0 1
```

The alternate scan *F*\ is defined analogously, with the split occurring along the **last** axis, and with the supernumerary axis placed **last**. For example:

```
    ×\3 3⍴⍳9
0   0   0
3  12  60
6  42 336
```

TIL, CUT, AND BIND

The operators discussed in this section are not essential to the rationalization program, but they can be used to further exploit, and therefore to demonstrate, the power of the essential facilities. In what follows, we will use the fact that the **con**

operator ⍤ is (in the dyadic use of the resulting derived function) a commutation, that is:

```
α F⍤ ω ← ω F α
```

The til operator, defined by

```
α F}G ω ↔ (G ω) F α
  F}G ω ↔ (G ω) F ω
```

can be used in a variety of ways. For example:

```
    F←÷⍤¯1
    G←⍵
    A←2 4 5
    B←10 2 4
    A F}G B                    Reverse B and divide by A
2 0.5 2

    A F}G⍤ B                   Reverse A and divide by B
0.5 2 0.5                      (a "left composition" in which G
                               applies only to the left argument)

    A F⍤}G B                   Right composition
0.5 2 0 5
```

If *F* is commutative, then right composition can be obtained more simply by the expression *F}G*. For example:

```
    A +}÷ B
2.1 4.5 5.25

    +}÷/C←2 4 5 6
2.23846

    +}÷\C
2 2.25 2.2381 2.23846
```

The case +}÷/*C* is equivalent to 2+÷4+÷5+÷6, and is called the continued fraction represented by *C*; the case +}÷*C* yields the "convergents" to the continued fraction.

The most important property of the til operator remains to be stated, namely:

```
F}G}H ω ↔ (Gω) F (Hω)
```

In other words, *F}G}H ω* applies a dyadic function *F* to the results of the monadic functions *G* and *H*. For example:

```
F←'ω*2'∇∘⍤0
```

```
    G←'ω*3'∇∘⍤0

    +}F}G} 1 2 3
2 12 36

    -}F}G 1 2 3
0 ¯4 ¯18

    I←'ω=⌊ω'∇∘⍤0
    P←'ω>0'∇∘⍤0
    N←¯1 ¯.5 0 .5 1

    ∧}I}P N
0 0 0 0 1
    ∨}I}P N
1 0 1 1 1
    ≠}I}P N
1 0 1 1 0
```

Often denoted by f+g without indicating that + is an operator rather than a function

The **cut** operator takes an integer left argument and a function right argument and applies it to pieces cut from the right argument of the derived function, putting the supernumerary axes contributed by the set of pieces in first place. The type of cut is determined by the left argument of the operator, and the points of cutting by the left argument of the derived function. The cuts are along the leading axis. For example:

```
    T←'ABCDEFGHIJ'
    B←0 0 1 0 0 0 1 1 0 0

    B 1⍤< T
|AB| |DEF| || |IJ|

    B ¯1⍤< T
|AB| |CDEF| |G| |HIJ|

    B 2⍤< T
|AB| |DEF| || |IJ|

    B ¯2⍤< T
|ABC| |DEFG| |H| |IJ|

    B 1⍤(+/) ⍳10
1 12 0 17

    B ¯1⍤(+/) ⍳10
1 14 6 24

    M←⍉2 10⍴'ABCDEFGHIJK'
    M
AK
```

A 1-cut begins pieces at 1's and excludes cut points

A ¯1-cut begins at 1's and includes cut points

(2 ¯2)-cuts end at 1's

```
BA
CB
DC
ED
FE
GF
HG
IH
JI
    B 1⍤< M
|AK| |DC| || |IH|
|BA| |ED|    |JI|
     |FE|

    S←'NOW IT IS'

    1⍤< ' ',S                    Monadic (1 ¯1)-cuts use the
|NOW| |IT| |IS|                  leading element as a delimiter

    ¯1⍤< ' ',S
| NOW| | IT| | IS|

    2⍤< S,' '                    Monadic (2 ¯2)-cuts use the last
|NOW| |IT| |IS|                  element as delimiter

    ¯2⍤< S,' '
|NOW | |IT | |IS |
```

The **general** cut $0 \overset{\circ}{} F$ has left rank 2 and infinite right rank, so that a left argument of rank greater than 2 produces a set of cuts. The first row of the left argument specifies the beginning point, and the magnitudes of the last row specify the size of the window, with a negative value producing a reversal along the corresponding axis. For example (using the identity function ⊢):

```
    A←>1 1⊃2 2
    A
1 1
2 2
    A 0⍤⊢ M←4 4⍴⍳16
 5  6
 9 10
    (>1 1⊃¯2 2) 0⍤< M
| 9 10|
| 5  6|
```

The 3-cut $\alpha \; 3 \overset{\circ}{} F \; \omega$ provides a tesselation, with the movement specified by α[0;], and the window again by α[1;]. For example (with □*PS* set to -1 1 3 3):

```
    A←2 2⍴2
```

```
    A 3⍤<M
 ┌─────┐ ┌─────┐
 │ 0 1 │ │ 2 3 │
 │ 4 5 │ │ 6 7 │
 └─────┘ └─────┘

 ┌───────┐ ┌────────┐
 │  8  9 │ │ 10  11 │
 │ 12 13 │ │ 14  15 │
 └───────┘ └────────┘

    A←>1 1⊃2 2
    ⍴R←A 3⍤⊢ M
3 3 2 2
    1{R
 4  5
 8  9

 5  6
 9 10

 6  7
10 11
```

In the expression α *F.K* ω, the binding operator causes *F* to be applied to the arguments with their first *K* axes "bound together" (and necessarily agreeing), and with the remaining **free** axes (exclusive of those demanded by cells as determined by the ranks of *F*) forming an outer product. The main point is best illustrated by a scalar function *F* in which no axes are demanded by cells:

```
    A←10×B←2 3⍴⍳6

    A +. 2 B
 0 11 22
33 44 55

    A +. 1 B
 0  1  2
10 11 12
20 21 22

33 34 35
43 44 45
53 54 55

    ⍴R←A +. 0 B
2 3 2 3
```

```
    R≡A∘.+B
1
```

The monadic case of *F.K* provides *K* successive reductions by *F* over the leading axes. For example:

```
    A←2 3 4 5⍴⍳120
    +. 2 A
300 306 312 318 324
330 336 342 348 354
360 366 372 378 384
390 396 402 408 414
```

For many functions (particularly for non-scalar functions) it is important to note that the reductions are carried out in the manner illustrated below:

F. 3 ω ↔ F⌿ F⌿⍤¯1 F⌿⍤¯2 ω

APL2 VERSUS A COMPARABLE SUBSET

This section presents comparisons between the facilities of APL2 [2] and a roughly equivalent subset of the facilities defined in [1]. The restricted subset (to be referred to as RS) embraces:

1. The functions <, >, and ≡ (enclose, disclose, and match), and the operators ⍤, ⍥, and ⍨ as defined (for function arguments only) in [4].

2. Argument ranks assigned to all primitive functions, and a rank operator (used in the form *F⍤R*) that assigns rank *R* to a derived function.

3. A single rule, common to all functions, for extending them to arguments of higher rank; the result shape is the catenation of the **outer shape** (exclusive of the **cell** shapes demanded by the ranks of the function ranks) with the common result shape produced by applying the function to individual cells.

4. The functions link (⊃) and from ({), including the dyadic case of *I⍨{* (which provides a merge equivalent to indexed assignment, as defined in [1]).

5. The transpose operator, which applies a function *F* after effecting a transpose; *F⍥I* moves axes *I* to the rear, and *I⍥F* to the front.

6. Use of the form *A←EXP* to assign a name to a function or operator as well as to a variable, and of the function definition operator ∇ which applies to vectors representing (in direct definition form) the monadic and dyadic cases of a function.

The functions from ({) and merge (*I*¨{), and the rank and transpose operators obviate all uses of brackets and semicolons. The **discussion** of RS will therefore be based on the assumption that their use is excluded, even though, for reasons of compatibility, they will continue to function as in conventional APL.

The comparisons made in this section are more theoretical than practical, concentrating on general questions such as implications for syntax, function classifications, and domains of operators. More practical comparisons of convenience, ease of learning, etc. are better made by the reader, either by drawing from his own applications, or from definitions and examples given in the APL2 manual [2] and the companion Introduction manual [3] as well as from the earlier sections of this paper, and its companion [1].

For example, one might take examples of strands and of heterogeneous arrays from the APL2 publications and write them completely in RS, and take function definitions such as *R←,*¨2 and write them completely in APL2.

Syntax. A useful measure of the complexity of a set of syntax rules is the required **scope**, that is, the maximum number of elements in the execution stack which must either be involved in the determination of the next action to be taken, or be used in that action. This measure is meaningful only on the assumption that each element is a normal operator, function, or variable, that is, an element does not incorporate further information relevant to another element or elements.

We will further assume that parentheses are handled by recursion on any segment enclosed by matching parentheses, and that parentheses are therefore not considered in assessing scope. However, this exclusion can not apply to an "unusual" use of parentheses, such as the usage *(C\[A]V)←X* in APL2, in which the result of the parenthesized expression is a "qualified" name.

Under the foregoing assumptions, the scope in RS is 4. In APL2:

1. Strands impose infinite scope.

2. The scope requirements of bracketed axis operators remain, or are further exaggerated by the introduction of semicolons within them.

3. Although the index function (denoted by the composite symbol "squad") shows some similarity to { in RS, it is not clear that an expression such as (1 2⊃?3 4⍴2){⍺3 could be conveniently expressed without the use of bracket indexing, and it may be reasonable to assume that bracket indexing is not superseded by squad.

4. Although the use of bracket indexing in indexed assignment may be partly (or even wholly) superseded by constructs such as (1 1⌷*V*)←*C*, this special use of parentheses to enclose a "qualified name" would appear to have the same implications for syntax as the use of bracket-index assignment.

5. The introduction of **ambiguous symbols** (that is /, ⌿, \, and ⍀) to denote both a monadic operator and a dyadic function, would appear to complicate the syntax. Thus, although the expressions +/1 2 and //1 2 are analyzed similarly, the expressions ⍴+/1 2 and ⍴//1 2 are not.

Function Classes. In conventional APL, functions are, explicitly or implicitly, divided into a number of overlapping classes, and the behaviour of a function depends, in significant ways, on the classes to which it belongs. Thus:

1. Scalar functions are all of rank 0, but, more significantly, they alone are in the domain of the operators /, ⌿, \, ⍀, and . (dot).

2. "Bracket" functions, which include certain derived functions such as +/ and +⍀ as well as the functions ⌽ and ⍉ are in the domain of the "bracket axis specification".

3. Some functions (such as ~, ⍋, ⍒, ⊥, ⊤, and ⍎) are fixed in valence, and produce syntax errors when used inappropriately.

4. Defined functions are included in class 3 above, but excluded from classes 1 and 2.

5. Derived functions are excluded from class 1, and some, but not all, are excluded from class 2.

In RS all such class distinctions are removed. However, each function is assigned a set of ranks, the ranks for all primitives of conventional APL being 0, 1, 2, or unbounded.

In APL2, some of these distinctions are removed in that operators such as reduction and inner product apply to non-scalar and defined and derived functions.

However, class 2 is not only retained, but extended (to scalar functions, to ↑, etc.), and class 3 (fixed valence) is also retained and extended, as evidenced by entries under "monadic" and "dyadic" in the glossary, by references to dyadic functions in the definitions of certain operators, and by the fact that the new bracket axis operator (distinct from the conventional "bracket axis specification" referred to in the definition of class 2) produces a function which is specifically monadic or dyadic.

Two further classes are introduced in APL2, called **pervasive** and **selection**: members of the former applying down through any structure of enclosures, and members of the latter being permitted to the left of assignment. No provision appears to be made for making a defined or derived function a member of either of these classes.

Depth Functions. APL2 provides two significant facilities which apply at "depth" in the enclosure structure of an argument, the dyadic **pick** function, and the pervasive functions. RS provides no primitive corresponding to **pick**; it could be defined recursively by:

```
PICK←''∇('→2+0=ρα'⊃'(>0{α){(1↓α)Δω'⊃'ω')
```

If frequent use demands greater convenience and efficiency, the pick function could be incorporated as a primitive. If so, the dyadic case of *I¨PICK* should provide the merge corresponding to the "selective specification" of the form *(L⊃R)←X* provided in APL2.

Since pervasiveness is a property assigned to functions, it would, in the framework of RS, be provided by an operator. Such an operator could be applied to any function (defined or derived as well as primitive) and, if defined to be dyadic, could provide greater variety. Thus a variable *P* in *F OP P* could provide application at any depth from the top, or any distance from the leaves; a **function** *P* (that is, a proposition) could specify the condition at the level where *F* is to apply.

Array Formation. If each essential space in an expression is counted as a character, then the link function and strand notation used to form non-simple vectors from simple elements require expressions of nearly identical length. For example:

Strand **Link**

```
(1 2)(3 4)(5 6)7 8      1 2⊃3 4⊃5 6⊃7 8
((1 2)3 4)((5 6)7 8)    (1 2⊃3 4)⊃<5 6⊃7 8
A B C D E F G H         A⊃B⊃C⊃D⊃E⊃F⊃G⊃H
(A B)(C D)(E F)(G H)    (A⊃B)⊃(C⊃D)⊃(E⊃F)⊃<G⊃H
A(B(C(D(E F))))         A⊃<B⊃<C⊃<D⊃<E⊃F
```

In conventional APL, the precedence of operators obviated parentheses around operator expressions, as in *A*+.×*B*, and in RS the precedence and scope rules for operators extend this convenience to expressions such as *F*⍤*A B*, regardless of whether *A* is a function or a variable. APL2 adopts the same rule for operators, but because strands take precedence over everything, an operator expression with a variable right argument must be parenthesized.

In conventional APL, two vector constants are never juxtaposed. In both RS and APL2, however, juxtaposition may occur in the case of any operator having a variable right argument (necessarily a user-defined operator in the case of APL2). In that event, the divide between the constants must be indicated by some delimiter. Thus:

```
F⍤1 2 (3 4 5)
F⍤(1 2) 3 4 5
F⍤1 2 A← 3 4 5
(F⍤1 2) 3 4 5
```

In RS, any one of these may be used, the first two being preferred because they clearly indicate the purpose of the parentheses, and the third being used if there are extraneous reasons for naming the right argument. In APL2, strands limit the choice to the fourth.

Retention of a valence classification for functions may have some implication for strands. Thus, if *F* is strictly monadic, the expression *A*~*B* might be considered identical to *A*(~*B*), since the strictly monadic function would presumably be executed first.

Although the valence classification is made in conventional APL (with *A*~*B* yielding a syntax error rather than a domain error) the valence classification has no serious consequences, because the juxtaposition of the two variables occasions an immediate error. Paradoxically, the function ~ is actually treated as ambivalent in the sense that the name *A* in the expression *A*~*B* is evaluated **before** the syntax error is reported, as may be seen by making *A* a shared variable (perhaps □) or a niladic function that specifies a global variable.

Domains of Operators. In conventional APL, the domains of operators are defined by one general rule (which excludes all defined functions), and by specific lists, the list of "scalar

primitives" for all but the bracket axis operator, and a second list (which includes some derived functions) for it.

Moreover, the domains of the derived functions produced are limited in ways which depend upon the arguments of the operators, depending upon the normal definition of the argument (as in ÷/4 0), or upon an "attribute" that it may possess (as in ⍲/⍳0).

In RS, the domains of operators are unrestricted, although the domains of the resulting derived functions continue to be restricted in ways analogous to those in conventional APL. Thus the domain of *F⍤G* will be restricted by the domains of *G*, *F*, and *G*⍣¯1 (that is *G* inverse). In particular, the domain of *F⍤G* is empty if the domain of *G*⍣¯1 is empty.

In APL2, the domains of most operators are unrestricted. However, the list of functions subject to (bracket) axis specification is not only maintained (as required for compatibility, even in RS), but is extended, as in *A*+[2]*B*, and in *A*↑[2]*B*. In the latter case of ↑, the "default" axis used when the axis specification is absent is not the final axis used for similar cases, but **all** axes.

Although the bracket notation used to represent it in conventional APL tends to obscure the matter, the axis operator is, in effect, **dyadic**, applying to a function "left argument" and a variable "right argument". In APL2, the bracket axis operator is defined to be **monadic**, seeming to imply that there is an indeterminate number of such operators, such as [1;1], and [1;2], etc.

Prototypes. Prototypes in APL2 are an extension of the notion (implicit in the difference in the results of 1↑⍳0 and 1↑'') that an array *A* incorporates some information other than its shape and the values of each of its elements.

In RS, compatibility with conventional APL is maintained, but no new distinctions are introduced. Specifically, 1↑0⍴1*J*1 and 1↑0⍴(1 2 3⊃'*AB*') both yield 0.

This is a fundamental difference that cannot be eliminated (as can the lack of a pervasive class of functions) by the introduction of an appropriate operator.

Heterogeneous Arrays. RS does not include the heterogeneous arrays of APL2, and the production of equivalent constructs requires greater use of enclosure. However, the structure of RS does not preclude their introduction.

Permissive Enclose. The monadic enclose functions defined in RS (<) and in APL2 (⊂) differ in one respect: if *S* is a simple scalar, then *S*≡⊂*S*, but ~*S*≡<*S*. Although < can therefore produce some structures not producible by ⊂, the differences between them (in the contexts of the respective systems) cannot, in most cases, be discerned.

The function { in RS provides an example where the difference is significant: because a doubly enclosed element of the left argument provides **complementary** indexing, the value of <<*K* must be distinguishable from <*K*, even in the case of a scalar *K*.

A similar situation would arise in the definition of an indexing function *F* in which *A F I* selects elements from *A* and incorporates them in the same "enclosure structure" as *I*. However, the difference would be discernible only in the case of some scalar element of *I* **and** a non-simple argument *A*.

General Remarks. The following general contrasts may be noted:

1. The definitions of function rank and of the disposition of result axes in RS permit the handling of constructs that would be handled by explicit encloses in APL2, and APL2 provides more facilities (such as pervasiveness) for operating "at depth".

2. RS provides facilities which obviate the use of brackets and semicolons in both indexing and in operators; although alternative indexing functions in APL2 could supersede bracket indexing, the use of brackets as operators is not only retained, but extended.

3. By assigning specified ranks to all functions, and by declaring all functions to be ambivalent, RS subsumes all in a single class, eliminating distinctions in the treatments of the classes (such as defined, derived, mixed) which occur in conventional APL.

   Although APL2 removes the confining distinction between primitives and defined and derived functions in the application of operators, it retains other function class distinctions of conventional APL, and introduces two further classes, pervasiveness, and "selectivity".

4. APL2 retains and extends the use of indexed assignment, whereas RS provides equivalent merge functions based upon the indexing function.

5. RS restricts the scope of syntax rules to 4, whereas APL2 makes no corresponding restriction.

REFERENCES

1. Iverson, K.E., **Rationalizing APL.**

2. APL2 Manual.

3. McGrew, J., **An Introduction to APL2.**

4. Bernecky and Iverson, 1980.